Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion ...

Review: Components of a Computer

Processor

- Control
- Datapath
- PC
- Registers

Memory

- Data
- Program

IO

Input

Output

Review: RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>op</th>
<th>imm</th>
<th>rd</th>
<th>rs1</th>
<th>rs2</th>
<th>funct7</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J-Format for Jump Instructions (JAL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>JAL</td>
</tr>
</tbody>
</table>

- JAL saves PC+4 in register rd (the return address)
- Assembler "j" jump is pseudo instruction, uses JAL but sets rd=n0 to discard return address
- Set PC = PC + offset (PC-relative jump: offset = signed immediate * 2)
- Target somewhere within $2^{16}$ locations, 2 bytes apart
- $2^{20}$ 32-bit instructions, 32 bytes
- Immediate encoded optimized similarly to branch instruction to reduce hardware cost
Uses of JAL

# j pseudo-instruction
# Label = jal a0; Label # Discard return address
# Call function within 218 instructions of PC jal ra, FuncName

JALR Instruction (I-Format)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>hi20</td>
<td>20-12</td>
</tr>
<tr>
<td>lo12</td>
<td>11-0</td>
</tr>
</tbody>
</table>

j Label = jal x0, Label # Discard return address
jal ra, FuncName

• JALR rd, rs, immediate
  – Writes PC+4 to rd (return address)
  – Sets PC = rs + immediate (12 bit, sign extended)
  – Uses same immediate as arithmetic and loads
• Unlike branches, no multiplication by two before adding to rs to form the new PC
• Byte offset NOT halfword offset as in branches and JAL

Uses of JALR

# ret and jr pseudo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>
jal x0, x1, <lo12bits>

Outline

• Review Instruction Formats + J/JAL
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
• Paper and pencil example (unsigned):
  \[
  \begin{array}{c}
  \text{Multiplicand} \quad 1000 \\
  \text{Multiplier} \quad \times 1001 \\
  \hline
  1000 \\
  \hline
  0000
  \end{array}
  \]
  \[
  \begin{array}{c}
  0000
  \end{array}
  \]
  \[
  \begin{array}{c}
  72
  \end{array}
  \]

• \(m \text{ bits} \times n \text{ bits} = m + n \text{ bit product}\)

• In RISC-V, we multiply registers, so:
  – 32-bit value \(\times\) 32-bit value = 64-bit value
  Syntax of Multiplication (signed):
  – MUL performs an 32-bit\(\times\)32-bit multiplication and places the lower 32 bits in the destination register.
  – MULH performs the same multiplication but returns the upper 32 bits of the full 2\(\times\)32-bit product.
  – If 64-bit product is required, then the recommended code sequence is:
    MULH \(dth, r1, r2\)
    MUL \(rdl, r1, r2\)
    (source register specifiers must be in same order and \(dth\) cannot be the same as \(r1\) or \(r2\))
Integer Multiplication (3/3)

• Example:
  - in C:  \( a = b \times c \); \( a \) should be declared long long
  - in RISC-V:
    * let \( b \) be 3; let \( c \) be 2; and let \( a \) be 10 and 1 (since it may be up to 64 bits)
    * \( \text{mulh t0,t2,t3} \) # upper half of product into \$s0
    * \( \text{mul t1,t2,t3} \) # lower half of product into \$s1

Integer Multiplication (3/3)

• Example:
  - in C:  \( a = b \times c \); \( a \) should be declared long long
  - in RISC-V:
    * let \( b \) be 3; let \( c \) be 2; and let \( a \) be 10 and 1 (since it may be up to 64 bits)
    * \( \text{mulh t0,t2,t3} \) # upper half of product into \$s0
    * \( \text{mul t1,t2,t3} \) # lower half of product into \$s1

Integer Multiplication (3/3)

• Example:
  - in C:  \( a = b \times c \); \( a \) should be declared long long
  - in RISC-V:
    * let \( b \) be 3; let \( c \) be 2; and let \( a \) be 10 and 1 (since it may be up to 64 bits)
    * \( \text{mulh t0,t2,t3} \) # upper half of product into \$s0
    * \( \text{mul t1,t2,t3} \) # lower half of product into \$s1

Integer Division (1/2)

• Paper and pencil example (unsigned):
  \[
  \begin{array}{c|c}
  \text{Dividend} & \text{Quotient} \\
  \hline
  10001010 & \quad \quad \quad \quad \quad \quad \quad \\
  \end{array}
  \]

  * Dividend = Quotient \times \text{Divisor} + \text{Remainder}

Integer Division (1/2)

• Paper and pencil example (unsigned):
  \[
  \begin{array}{c|c}
  \text{Dividend} & \text{Quotient} \\
  \hline
  10001010 & \quad \quad \quad \quad \quad \quad \quad \\
  1000 & \quad \quad \quad \quad \quad \quad \quad \\
  \end{array}
  \]

  * Dividend = Quotient \times \text{Divisor} + \text{Remainder}

Integer Division (1/2)

• Paper and pencil example (unsigned):
  \[
  \begin{array}{c|c}
  \text{Dividend} & \text{Quotient} \\
  \hline
  10001010 & \quad \quad \quad \quad \quad \quad \quad \\
  1000 & \quad \quad \quad \quad \quad \quad \quad \\
  \end{array}
  \]

  * Dividend = Quotient \times \text{Divisor} + \text{Remainder}

• Paper and pencil example (unsigned):
  \[
  \begin{array}{c|c}
  \text{Dividend} & \text{Quotient} \\
  \hline
  10001010 & \quad \quad \quad \quad \quad \quad \quad \\
  1000 & \quad \quad \quad \quad \quad \quad \quad \\
  \end{array}
  \]

  * Dividend = Quotient \times \text{Divisor} + \text{Remainder}
Integer Division (1/2)

- Paper and pencil example (unsigned):
  
<table>
<thead>
<tr>
<th>Dividend</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>1001010</td>
<td>1010</td>
</tr>
</tbody>
</table>

- Dividend = Quotient \times Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):
  
<table>
<thead>
<tr>
<th>Dividend</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>1001010</td>
<td>1010</td>
</tr>
</tbody>
</table>

- Dividend = Quotient \times Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):
  
<table>
<thead>
<tr>
<th>Dividend</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>1001010</td>
<td>1010</td>
</tr>
</tbody>
</table>

- Dividend = Quotient \times Divisor + Remainder

Integer Division (2/2)

- Syntax of Division (signed):
  - DIV performs signed integer division of 32 bits by 32 bits. REM provides the remainder of the corresponding division operation.
  - If the quotient and remainder are required from the same division, the recommended code sequence is:
    - DIV rdq, rs1, rs2
    - REM rdr, rs1, rs2
    (rdq cannot be the same as rs1 or rs2)

- Implements C division (/) and modulo (%)

- Example in C:
  
  ```
  a = c / d;  b = c % d;
  ```

- In MISC:
  
  ```
  add c, t1, t2, d1, t3
  ```

Peer Question

Which of the following places the address of LOOP in t1?

1) la t2, LOOP
   lw t1, 0(t2)
   1 2 3

2) jal LOOP
   LOOP: add t1, x0, x0
   1 2

3) la t1, LOOP
   1 2 3
Peer Question

Which of the following places the address of LOOP in t1?

1) la t2, LOOP
   lw t1, 0(t2)
2) jal LOOP
   LOOP: add t1, ra, x0
3) la t1, LOOP

Levels of Representation/Interpretation

Language Execution Continuum

- **Interpreter** is a program that executes other programs
- Language translation gives us another option
  - In general, we interpret a high-level language when efficiency is not critical and translate to a lower-level language to increase performance

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
  - Assembler
  - Linker
  - Loader
  - And in Conclusion ...
Interpretation

- For example, consider a Python program `foo.py`
- Python interpreter is just a program that reads a python program and performs the functions of that python program

Interpretation

- WHY interpret machine language in software?
- VENUS (Lab #3), Project #1: RISC-V simulator useful for learning/debugging
- E.g., Apple Macintosh conversion
  - Switched from Motorola 68000 instruction architecture to PowerPC (before x86)
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)

Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., VENUS, Project #1)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine

Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems
- Translation/compilation helps "hide" the program "source" from the users:
  - One model for creating value in the marketplace (e.g., Microsoft keeps all their source code secret)
  - Alternative model, "open source", creates value by publishing the source code and fostering a community of developers.

Steps in Compiling a C Program

Compiler

- Input: High-Level Language Code (e.g., `foo.c`)
- Output: Assembly Language Code (e.g., `foo.s` for RISC-V)
- Note: Output may contain pseudo-instructions
  - Pseudo-instructions: instructions that assembler understands but not in machine
- For example (move t2 to t1):
  - `mv t1,t2 -> addi t1,t2,0`
Outline

• Review Instruction Formats + J/JAL
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion …

Where Are We Now?

Assembler

• Input: Assembly Language Code [includes pseudo ops]
  (e.g., foo.s for RISC-V)
• Output: Object Code, information tables (true assembly only)
  (e.g., foo.o for RISC-V)
• Reads and Uses Directives
• Replace Pseudo-instructions
• Produce Machine Language
• Creates Object File

Assembler Directives
(See RISCV Reader, Chapter 3)

• Give directions to assembler, but do not produce machine instructions
  .text: Subsequent items put in user text segment (machine code)
  .data: Subsequent items put in user data segment (binary rep of data in source file)
  .globl sym: declares sym global and can be referenced from other files
  .string str: Store the string str in memory and null-terminate it
  .word w1.wm: Store the n 32-bit quantities in successive memory words

Pseudo-instruction Replacement

• Assembler treats convenient variations of machine language instructions as if real instructions
  Pseudo: mv t0, t1
  Real: addi t0, t1, 0
  neg t0, t1, -1
  li t0, -1
  sub t0, zero, t1
  add t0, zero, imm
  xori t0, zero, zero
  beq t0, zero, loop
  la t0, str

Pseudo-instruction Replacement

• Assembler treats convenient variations of machine language instructions as if real instructions
  Pseudo: mv t0, t1
  Real: addi t0, t1, 0
  neg t0, t1, -1
  li t0, -1
  sub t0, zero, t1
  add t0, zero, imm
  xori t0, zero, zero
  beq t0, zero, loop
  la t0, str
  addi t0, str[11:0]
  addi t0, str[31:12]
  addi t0, str[11:0]

DON'T FORGET: sign-extended immediates

Branch immediates (count halfwords)

STATIC Addressing

PC Relative Addressing
Peer Instruction
Which of the following is a correct assembly language sequence for the pseudoinstruction: `la t1, FOO`?

A: `ori t1, 0xABCD0`  *Assume the address of FOO is 0xABCD0124*
B: `addi t1, 0x124`
C: `lui t1, 0xD0124`  `ori t1, 0xABC`
D: `lui t1, 0xABCD0`  `addi t1, 0x124`

*Assume the address of FOO is 0xABCD0124*

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches and Jumps?
  - PC-Relative (e.g., `beq`/`bne` and `jal`)
    - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch/jump over
  - So these can be handled

Producing Machine Language (2/3)

```
addi t2, zero, 9  # t2 = 9
L1:
slt t1, zero, t2  # 0 < t2? Set t1
beq t1, zero, L2  # NO! t2 <= 0; Go to L2
addi t2, t2, -1  # YES! t2 > 0; t2--
j  L1  # Go to L1
L2:
```

```
addi t2, zero, 9  # t2 = 9
bge zero, t2, L2  # 0 >= t2? Exit!
addi t2, t2, -1  # 0 < t2; t2--
j  L1  # Go to L1
L2:
```

```
addi t2, zero, 9  # t2 = 9
bge zero, t2, L2  # 0 >= t2? Exit!
addi t2, t2, -1  # 0 < t2; t2--
j  L1  # Go to L1
L2:
```

Producing Machine Language (3/3)

- What about PC-relative jumps (`jal`) and branches (`beq`, `bne`)?
  - `j offset` pseudo instruction expands to `JAL zero, offset`
  - Just count the number of instruction halfwords between target and jump to determine the offset: position-independent code (PIC)
- What about references to static data?
  - `la` gets broken up into `lui` and `addi` (use `auipc`/`addi` for PIC)
  - These require the full 32-bit address of the data
- These can’t be determined yet, so we create two tables ...

Symbol Table

- List of “items” in this file that may be used by other files
- What are they?
  - Labels: function calling
  - Data: anything in the .data section; variables which may be accessed across files
Relocation Table

- List of "items" whose address this file needs
  - Any absolute label jumped to: jal, jalr
    - Internal
    - External (including lib files)
    - Such as the la instruction
      - E.g., for jalr base register
    - Any piece of data in static section
    - Such as the la instruction
      - E.g., for jalr base register

Object File Format

- object file header: size and position of the other pieces of the object file
  - text segment: the machine code
  - data segment: binary representation of the static data in the source file
  - relocation information: identifies lines of code that need to be fixed up later
  - symbol table: list of this file's labels and static data that can be referenced
  - debugging information
    - A standard format is ELF (except MS)

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
  - Linker
  - Loader
  - And in Conclusion...

Where Are We Now?

Linker (1/3)

- Input: Object code files, information tables (e.g., .o, .so, .lib, .o for RISC-V)
- Output: Executable code (e.g., a.out for RISC-V)
- Combines several object (.o) files into a single executable ("linking")
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Linux sources > 20 Millions of codes
  - Old name "Link Editor" from editing the "links" in jump and link instructions

Linker (2/3)

...
**Linker (3/3)**

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve references
  - Go through Relocation Table; handle each entry
  - I.e., fill in all absolute addresses

**Four Types of Addresses**

- PC-Relative Addressing (*beq, bne, jal; auipc/addi*)
  - Never need to relocate (PIC: position independent code)
- Absolute Function Address (*auipc/jalr*)
  - Always relocate
- External Function Reference (*auipc/jalr*)
  - Always relocate
- Static Data Reference (often *lui/addi*)
  - Always relocate

**Absolute Addresses in RISC-V**

- Which instructions need relocation editing?
  - J-format: jump/jump and link
    
    | xxxx | rd | jal |
    |------|----|-----|
  - I-, S- Format: Loads and stores to variables in static area, relative to global pointer
    
    | xxx | gp | rd | lw |
    |-----|----|----|----|
  - What about translation/handling?
    
    | xx | rs1 | gp | x | aw |
    |-----|-----|----|---|----|
  - PC-relative addressing preserved even if code moves
    
    | xx | rs1 | rs2 | x | beq |
    |-----|-----|-----|--|----|

**Resolving References (1/2)**

- Linker assumes first word of first text segment is at address 0x10000 for RV32.
  - (More later when we study “virtual memory”)
- Linker knows:
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates:
  - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

**Resolving References (2/2)**

- To resolve references:
  - Search for reference (data or label) in all “user” symbol tables
  - If not found, search library files (e.g., for *printf*)
  - Once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

**Administrivia**

- Midterm #1 in 1 week: September 26!
  - IN CLASS 11:30 AM Wheeler Auditorium + five Spill Rooms based on class login
  - Number Representations, C (pointers, arrays, strings), mapping C to RISC-V Assembly (and vice versa), NO MACHINE LANGUAGE ON THIS EXAM
  - Single double sided 8.5” x 11” cheat sheet + we give you the RISC-V Green Card
  - Review session Saturday, September 23: Time and place TBD
  - DSP students: please make sure we know about your special accommodations (contact TA Steven Hu)
Outline

• Review Instruction Formats + J/JAL
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...

Where Are We Now?

Loader Basics

• Input: Executable Code (e.g., a.out for RISC-V)
• Output: (program is run)
• Executable files are stored on disk
• When one is run, loader’s job is to load it into memory and start it running
  — In reality, loader is the operating system (OS)

Loader ... What Does It Do?

• Reads executable file’s header to determine size of text and data segments
• Creates new address space for program large enough to hold text and data segments, along with a stack segment
• Copies instructions + data from executable file into the new address space
• Copies arguments passed to the program onto the stack
• Initializes machine registers
  — Most registers cleared, but stack pointer assigned address of 1st free stack location
• Jumps to start-up routine that copies program’s arguments from stack to registers & sets the PC
  — If main routine returns, start-up routine terminates program with the exit system call

Peer Instruction

At what point in process are all the machine code bits determined for the following assembly instructions:

1) add x6, x7, x8
2) jal x1, fprintf

A: 1) & 2) After compilation
B: 1) After compilation, 2) After assembly
C: 1) After assembly, 2) After linking
D: 1) After assembly, 2) After loading
Assembled Hello.s: Linkable Hello.o

```
00000000 <main>
  0: f010113 addi sp,sp,-16
  4: 0012623 sw ra,12(sp)
  8: 0000537 lui a0,0x0
 c: 0000653 addi a0,a0,0
10: 00005b7 lui a1,0x0
14: 008893 addi a1,a1,0
18: 008987 addc ra,ra,0
1c: 0060067 jalr ra
20: 00c12b3 lw ra,12(sp)
24: 0010113 addi sp,sp,16
28: 0000513 addi a0,a0,0
2c: 0000867 jalr ra
```

Linked Hello.o: a.out

```
000101b0 <main>
101b0: f010113 addi sp,sp,-16
101b4: 0012623 sw ra,12(sp)
101b8: 00021537 lui a0,0x210
101bc: a105513 addi a0,a0,-1520 # &string1
101c0: 000215bf lui a1,0x21
101c4: a1c9593 addi a1,a1,-1508 # &string2
101c8: 288000ef jal ra,10450 # &printf
101cc: 00c12b3 lw ra,12(sp)
101d0: 0010113 addi sp,sp,16
101d4: 0000513 addi a0,a0,0
101d8: 0000867 jalr ra
```

LUI/ADDI Address Calculation in RISC-V

Target address of &string1 = 0x00020 A10
Instruction sequence LUI 0x00020 ADDI 0xA10 does not quite work because immediates in RISC-V are sign extended (and 0xA10 has a 1 in the high order bit) 0x00020 000 + 0xFF0FF A10 = 0x00020 A10 (off by 0x00001 000)
So we get the right address if we calculate it as follows:

```
(0x00020 000 + 0x00001 000) + 0xFFFFFF A10 = 0x00021 A10
```

What is 0xFFFFFF A10?
Two's complement of 0xFFFFFF A10 = 0x00000 0FF + 1 = 0x00001 000 LUI = 0x00020 000
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Static vs. Dynamically Linked Libraries

- What we’ve described is the traditional way: statically-linked approach
  - Library is now part of the executable, so if the library updates, we don’t get the fix (have to recompile if we have source)
  - Includes the entire library even if not all of it will be used
  - Executable is self-contained
- Alternative is dynamically linked libraries (DLL), common on Windows & UNIX platforms

Dynamically Linked Libraries

- Space/time issues
  - Storing a program requires less disk space
  - Sending a program requires less time
  - Executing two programs requires less memory (if they share a library) at runtime, there’s time overhead to do link
- Upgrades
  - Replacing one file (libXYZ.so) upgrades every program that uses library “XYZ”
    - Having the executable isn’t enough anymore

Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.

Dynamically Linked Libraries

- Prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  - Linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - Can be described as “linking at the machine code level”
  - This isn’t the only way to do it ...

Outline

- Review Instruction Formats + I/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion ...

And In Conclusion ...

- Compiler converts a single HLL file into a single assembly language file
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A .s file becomes a .o file
  - Does a pass to create addresses, binding internal global variables
- Linker combines several .o files and resolves absolute addresses
  - Enables separate compilation, libraries that need not be compiled, and modules remaining address
- Loader loads executable into memory and begins execution